1 Food Webs

1.1 Introduction

Food webs are collections of predation relationships between species living in the same habitat. As shown in Fig.1.1 such collections appear (and actually are) quite difficult to visualise and any reductionist approach misses the point of functionality description (not to mention the prediction of any future behaviour). The necessity of a comprehensive approach makes food webs a paramount example of a complex system (Havens, 1992; Solé and Montoya, 2001; Montoya and Solé, 2002; Stouffer *et al.*, 2005). Already in 1991, food webs had been described as a community of predators and parasites as "complex, but not hopelessly so" (Pimm *et al.*, 1991). More specifically, following a traditional approach (Cohen, 1977), ecologists distinguish among

- *community webs* defined by picking in the same habitat (or set of habitats) a group of species connected by their predation relationships;
- *sink webs* made by collecting all the prey eaten by one or more predators and recursively the prey of these prey;
- *source webs* made by collecting all the predators of one or more species and recursively the predators of such predators.

In any of these webs, prey or predators do not necessarily correspond to distinct species. Rather, the same species can appear more than once in different roles (i.e. different stages in the life cycle of an organism). Also, since cannibalism is present (both at the same and at different stages of growth of individuals) a single species can be both prey and predator.

Two problems arise: firstly, by considering the definition of community webs, it is clear that working out who eats whom can be in principle rather complicated and typically many years of field observation are necessary to spot particularly rare events. Secondly, when we list species in a given habitat (Willis and Yule, 1922) we are more inclined to spot very small differences between large animals (difference in colours of eagles or tigers), while we do not notice the differences between species of similar bacteria. As regards the first issue, there is no other solution than working hard to improve the quality of the data collected. Indeed, we now understand that scarcity of observation can lead to partial knowledge of predations, thereby underestimating the role of certain species in the environment (Martinez, 1991; Martinez *et al.*, 1999). For the second issue it is customary to reduce the observational bias by introducing the concept of *trophic species* (Memmott *et al.*, 2000). Those are a coarse grained version of species and they can be obtained by lumping together different organisms when they feed on the same prey and they are eaten by the same predators.

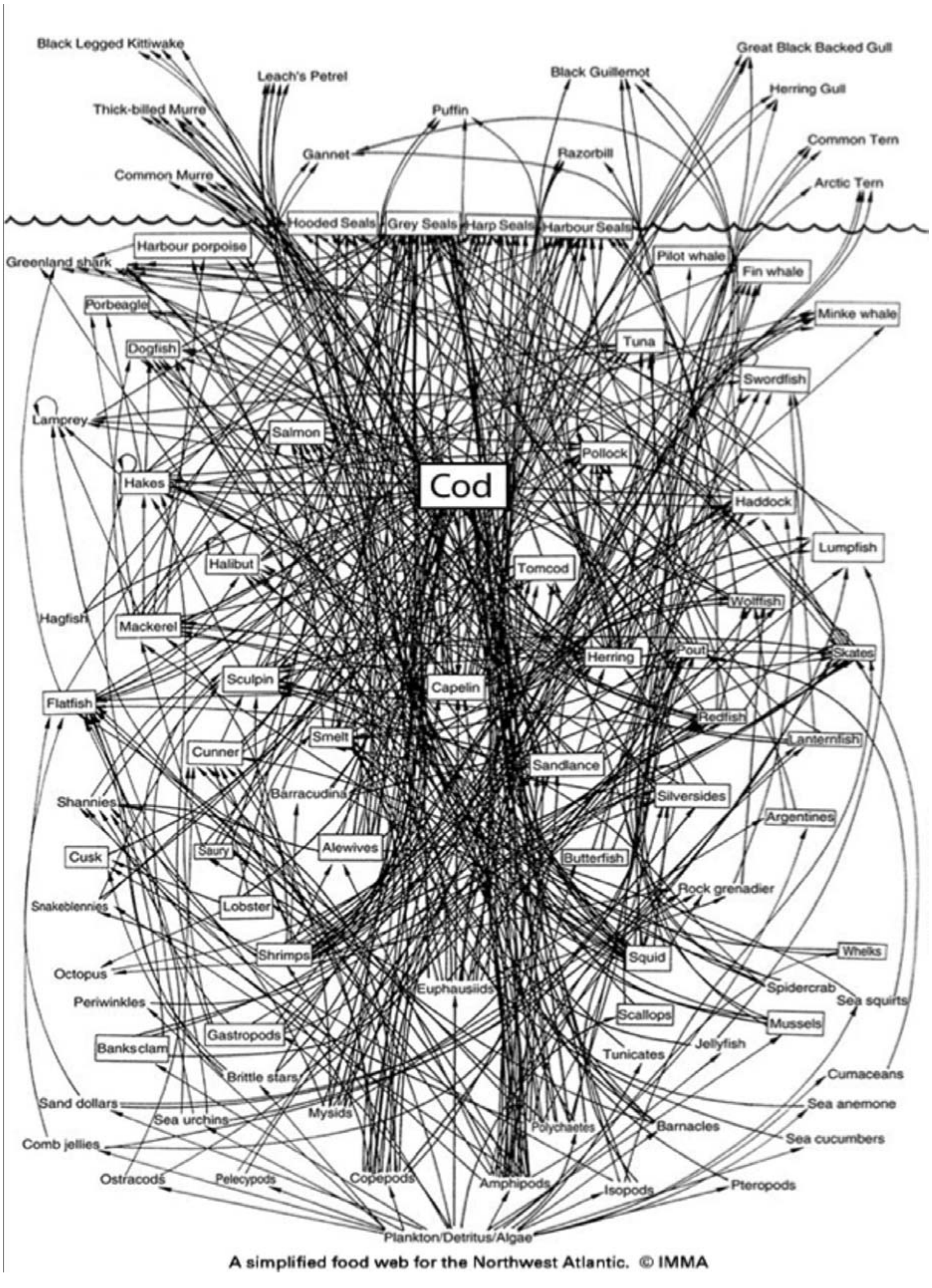

Fig. 1.1 An example of a partial marine food web.

Food webs are a typical example of a system where we find a natural characterisation in the form of a network. Directed edges (prey $\rightarrow$ predators) connect the various vertices (trophic species). Not surprisingly then, many ecological quantities have their counterpart in graph theory:

- number of trophic species S that are the number n of vertices (measure of a graph);
- number of links L that are the number m of edges (size of a graph);
- number of possible predations $\simeq S^2$ (with cannibalism $= S^2$, without $= S(S-1)$) corresponding to the size of a complete directed graph;
- connectance $C \simeq L/S^2$ of a given habitat, that is, the ratio between the number of edges present with respect to those that are possible (see previously);
- number of prey per species, that is the in-degree of the vertex and similarly the number of predators per species, that is the out-degree of the vertex;
- number of triangulations between species (Huxham *et al.*, 1996). In a directed graph it is related to the motifs structure, in a simplified form of an undirected graph with clustering.

Food webs, in particular, also show some particular behaviour that is rather uncommon in other fields: vertices can be naturally ordered according to a scale of levels (see Fig. 1.2). The vertices in the first level are species "predating" only water, minerals, and sunlight energy. In ecology those are known as "basal species" or primary producers. Species in the second level are those who predate on the first level (irrespective of the fact that they can also predate other species). This concept can be iterated and in general the *level* of one species is related to the minimum path to "basal species". In this way, we can cluster together trophic species into three simple classes:

- *basal species* B that have only predators;
- *top species* T that have only prey;
- *intermediate species* I that have both.

This allows us to define several quantities that we can use to describe the various food webs:

- Firstly, the proportion of such classes (Pimm *et al.*, 1991);
- Secondly, the proportion of the different links between the classes (BI, BT, II, IT);
- Finally, the ratio prey/predators (Cohen, 1977) $= (\#B + \#I)/(\#I + \#T)$.

We remind that codes, data and/or links for this chapter are available from **http://book.complexnetworks.net**

1.2 Data from EcoWeb and foodweb.org

A traditional source of data for species is given by the machine readable dataset EcoWeb,[1] presenting 181 small food webs. In all of them the number of species is

[1]https://dspace.rockefeller.edu/handle/10209/306

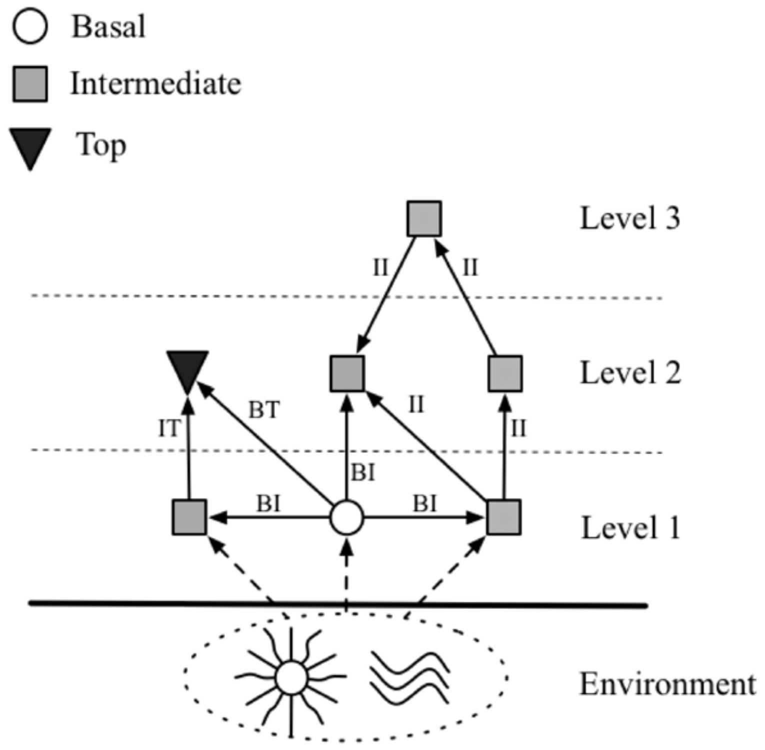

Fig. 1.2 Structure of a food web with the distinction in levels and classes of species.

rather small, even if in more recent datasets there is a somewhat larger number of species.

- St Martin Island (Goldwasser and Roughgarden, 1993) with 42 trophic species;
- St Marks Seagrass (Christian and Luczkovich, 1999) with 46 trophic species;
- Another grassland (Martinez *et al.*, 1999) with 63 trophic species;
- Silwood Park (Memmott *et al.*, 2000) with 81 trophic species;
- Ythan Estuary (without parasites) (Hall and Raffaelli, 1991) with 81 trophic species;
- Little Rock Lake web (Martinez, 1991) with 93 trophic species;
- Ythan Estuary (with parasites) (Huxham *et al.*, 1996) with 126 trophic species.

We have listed these food webs in the material attached to this book. Other datasets can be downloaded from:

- `http://vlado.fmf.uni-lj.si/pub/networks/data/bio/foodweb/foodweb.htm`
- `http://datadryad.org/resource/doi:10.5061/dryad.b8r5c`
- `https://networkdata.ics.uci.edu/`
- `http://globalwebdb.com/`

A good public resource where it is possible to download publications, software tools, and data is the basic PEaCE Lab web site at http://foodwebs.org.

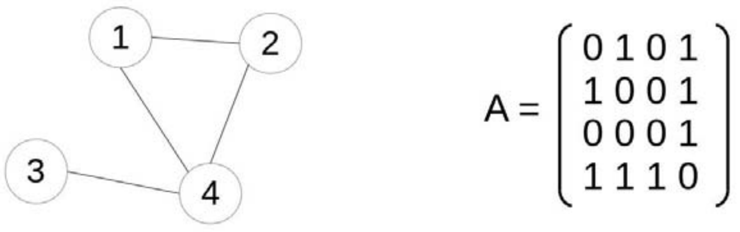

Fig. 1.3 A simple graph and its adjacency matrix.

1.3 Store and measure a graph: size, measure, and degree

1.3.1 The adjacency matrix

Graphs can be drawn (note that this can be done in many ways) either as a picture or by means of a mathematical representation known as an “adjacency matrix”, as shown in Fig.1.3. While the picture is useful for visualising immediately some properties of the graph (typically local ones), the matricial form is very useful for computing others (typically global). The adjacency matrix A is a square table of numbers ($n \times n$, where n is the number of vertices in the graph), where the elements in row i and column j are equal to 1 if we have an edge between vertex i and j, and equal to 0 otherwise. Such structures can be stored by computers and this representation turns out to be particularly useful when dealing with large datasets. It would be insufficient to believe that matrices are only useful as computer representations of graphs. Spectral properties (related to the matricial form) of graphs can reveal many properties, starting with community structure. We shall see these points in detail later in this book.

The simplest case of an undirected graph gives symmetric matrices A. If there is an edge between i and j there is also an edge in the opposite direction joining j and i, so that $a_{ij} = a_{ji}$. Instead, if there is a direction in the linkage we have an arc starting from the node i and pointing to the node j, but the opposite is not necessarily the case, in this way $a_{ij} \neq a_{ji}$ and the matrix is no longer symmetric. Undirected networks are the simplest form of graph we can deal with. Symmetric matrices indicate that we have only one kind of link connecting two vertices. In the case of the Internet this means that a cable connecting two servers can be used both for sending part of a mail from the first to the second and also for answering the mail from the second to the first. Asymmetric matrices describe situations where the path cannot be followed in reverse. If you put a link from your web page to the web page of your favourite football team or Hollywood actor, you will not necessarily (and probably seldom) receive a link back to your page. Oriented networks arise in a variety of situations. On the web in social networks where we have e-mail, “likes” on Facebook posts, and retweets on Twitter that are not symmetric; in finance where borrowing money is not lending it; in economics where we have trade between nations, and finally in biology where the predation relations in food webs are typically not reciprocal.

In this section we present the basic quantities used to describe a graph (some have already been mentioned), together with an example of the Python code.

Read the adjacency matrix
Starting from this formulation it is relatively easy to cast it in Python code. We can either start with the matrix representation of the graph or with the direct representation of nodes and edges from the Networkx library. In the first case, we will first represent the graph through the adjacency matrix using the basic list data type (in this case list of lists) in the following way:

```
adjacency_matrix=[
                  [0,1,0,1],
                  [1,0,1,1],
                  [0,1,0,0],
                  [1,1,0,0]
                  ]
```

The basic Python statement for iterative cycles is a little bit different from the usual programming languages, like Fortran, C/C++, and the like. The iteration is supposed to run over a list and, for example, in a simple case of an index i running from 1 to 5 the syntax will be:

```
for i in [1,2,3,4,5]:
    print i

#OUTPUT
1
2
3
4
5
```

We can browse the matrix rows using the "for" Python statement:

```
for row in adjacency_matrix:
    print row

#OUTPUT
[0, 1, 0, 1]
[1, 0, 1, 1]
[0, 1, 0, 0]
[1, 1, 0, 0]
```

Note the indentation of the "print" statement, that is mandatory in Python, even if its length is not fixed.

To get each single matrix element we will nest another for cycling to extract each element of the rows:

```
for row in adjacency_matrix:
    for a_ij in row:
        print a_ij,
    print "\r"

#OUTPUT
0 1 0 1
1 0 1 1
0 1 0 0
1 1 0 0
```

The comma prevents the new line adding simply a space in the visualisation of the row, while the special character "\r" stands for a carriage return.

In the case of directed networks the adjacency matrix is not symmetric, like for a food web; if a non-zero element is present in row 2, column 3, this means there is an arc (directed edge) from node 2 towards node 3:

```
adjacency_matrix_directed=[
                    [0,1,0,1],
                    [0,0,1,0],
                    [0,0,0,1],
                    [0,0,0,0]
                    ]
```

1.3.2 Size, measure, connectance

The simplest scalar quantities defined in the title can be computed easily in the case of the various food webs. We recall them here:

- the number of species S, that in graph theory corresponds to the number of vertices n which is the *measure* of the graph;
- the number of predations L, that in graph theory corresponds to the number of edges m, which is the *size* of the graph;
- since in this case we can distinguish among the different nature of vertices (i.e. B,I,T), we can measure proportions of species and links between them;
- The connectance $C \simeq L/S^2$, corresponding to the density of the graph (actual edges present divided by the maximum possible number).

Basic statistics

```
#the number of species is the number of rows or columns of
#the adjacency matrix
num_species=len(adjacency_matrix_directed[0])

#the number of links or predations is the non zero elements
#of the adjacency matrix (this holds for directed graphs
num_predations=0
for i in range(num_species):
    for j in range(num_species):
        if adjacency_matrix_directed[i][j]!=0:
            num_predations=num_predations+1

#to check if a specie is a Basal (B), an Intermediate (I) or
#a Top (T) one  we have to check the presence of 1s both in
#the row and in the column of each specie
row_count=[0,0,0,0]
column_count=[0,0,0,0]
for i in range(num_species):
    for j in range(num_species):
        row_count[i]=row_count[i]+adjacency_matrix_directed[i][j]
        column_count[j]=column_count[j]+ \
        adjacency_matrix_directed[i][j]

number_B=0
number_I=0
number_T=0

for n in range(num_species):
    if row_count[n]==0:
        number_T+=1
        continue
    if column_count[n]==0:
        number_B+=1
        continue
    else:
        number_I+=1

print "number of species", num_species
print "number of predations", num_predations
print "classes Basal, Top, Intermediate: ",number_B,number_T,number_I
print "connectance", float(num_predations)/float(num_species**2)

#OUTPUT
number of species 4
number of predations 4
classes Basal, Top, Intermediate:  1 1 2
connectance 0.25
```

1.3.3 The degree

The simplest quantity that characterises the vertex is the number of its connections. This quantity is called the *degree*, sometimes (mostly by physicists) called "connectivity". The degree of a vertex indicates the connections of this vertex; the degree is thereby a "local" quantity (you need to inspect only one vertex to find its degree). In the following we shall see non-local measures of graphs, which involve two or more vertex neighbours, and also measures that are "global" and need an inspection of the whole system to be computed (i.e. betweenness). The frequency distribution of this quantity is traditionally used as a signature of "complexity", in the sense that the presence of long tails (or a scale-free distribution) is interpreted as a signature of long-range correlation in the system. When the graph is oriented we can distinguish between in-degree and out-degree. The former accounts for ingoing links (for example the energy we receive when eating another living organism), the latter accounts for outgoing links (as for example the hyperlinks we put on our web page to other pages we like). Once we have the matrix of the graph, the degree can easily be computed. If we want to know the degree k_i of the vertex i we simply sum the various elements a_{ij} on the various columns j, i.e.

$$k_i = \sum_{j=1,n} a_{ij}. \tag{1.1}$$

If the graph is oriented, the sum along the rows of the (non-symmetric) matrix A is different from the sum along the columns (that is not the case when the matrix is symmetric and the graph is not oriented). In one case we get the out-degree k_i^O of node i, while in the opposite we get the in-degree k_i^I of node i. In formulas,

$$k_i^I = \sum_{j=1,n} a_{ij}, \qquad k_i^O = \sum_{j=1,n} a_{ji}. \tag{1.2}$$

When the graph is weighted we can extend the previous definition, by distinguishing between the number of connections (degree) and the weighted degree or *strength* s , that is the sum of the relative weights of those links. Also in this case, we can use the matrix representation. Now every element a_{ij}^w takes the value of the weight between i and j, we have

$$s_i = \sum_{j=1,n} a_{ij}^w. \tag{1.3}$$

Typically in real situations there is a power-law relation between strength and degree (Barrat *et al.*, 2004)

Degree

With this matrix representation we can calculate the degree for a specific node (in this case the node "2"):

```
#for the undirected network
degree_node_2=0
for j in adjacency_matrix[1]:
    degree_node_2=degree_node_2+j
print "degree of node 2:",degree_node_2

#and for the directed case we already calculated the sum over
#the rows and columns for the adjacency_matrix_directed
out_degree_node_3=row_count[2]
in_degree_node_4=column_count[3]

print "out_degree node 3:",out_degree_node_3
print "in_degree node 4:",in_degree_node_4

#OUTPUT
degree of node 2: 3
out_degree node 3: 1
in_degree node 4: 2
```

Remember that the indices in Python data structures start from “0” and so the row “2” is marked as “1”.

Degree in Networkx
The equivalent procedure in Networkx will be:

```
import networkx as nx

#generate an empty graph
G=nx.Graph()

#define the nodes
G.add_node(1)
G.add_node(2)
G.add_node(3)
G.add_node(4)

#link the nodes
G.add_edge(1,2)
G.add_edge(1,4)
G.add_edge(2,3)
G.add_edge(2,4)

#degree of the node 2
print G.degree(2)

#OUTPUT
3
```

1.4 Degree sequence

When dealing with large networks we need only to coarse grain the information on connections by giving the degree sequence, that is the list of the various degrees in the graph. Such information can be summarised by making a histogram of the degree sequence (a typical and traditional statistical analysis done for complex networks that serves as a benchmark to describe the suitability of various models of network growth). Please note that while we can associate a degree sequence to any graph, obviously not all sequences of numbers can produce a graph (see also Sec. 6.4). For example, the sum of all the degrees in an undirected graph must be an even number (we are counting every edge twice, then the sum of the elements in the degree sequence gives $2E$, where E is the total number of edges). As a consequence, any degree sequence whose sum is odd, cannot form a graph. Furthermore even if the sum of elements in the degree sequence is even, most configurations are impossible (imagine a degree larger than the number of vertices present). Empirically, whenever graphs are made from a large number of vertices, it becomes more and more difficult to check if a given degree sequence is actually describing a graph or not.

We shall see more on these topics in Chapter 6, for the moment let us focus only on the passage from the graph to the degree sequence. A simple way to obtain the degree sequence starting from the previous Python formulation is to generalise the code in order to compute the degree for each row, as follows:

Degree sequence

```
degree_sequence=[]
for row in range(len(adjacency_matrix)):
    degree=0
    for j in adjacency_matrix[row]:
        degree=degree+j
    degree_sequence.append(degree)

print degree_sequence
```

and the output will be:

```
#OUTPUT
[2, 3, 1, 2]
```

1.4.1 Plotting the degree sequence, histograms

As mentioned previously, when the network is large, we want a single plot or image that might help us in describing the graph. A histogram is the best choice for that purpose and it is important to learn how to draw these objects from analysis of the raw data. In practice, we must count how many times we have a vertex whose degree is $1, 2$, etc. This number is plotted against the degree values as in Fig. 1.4.

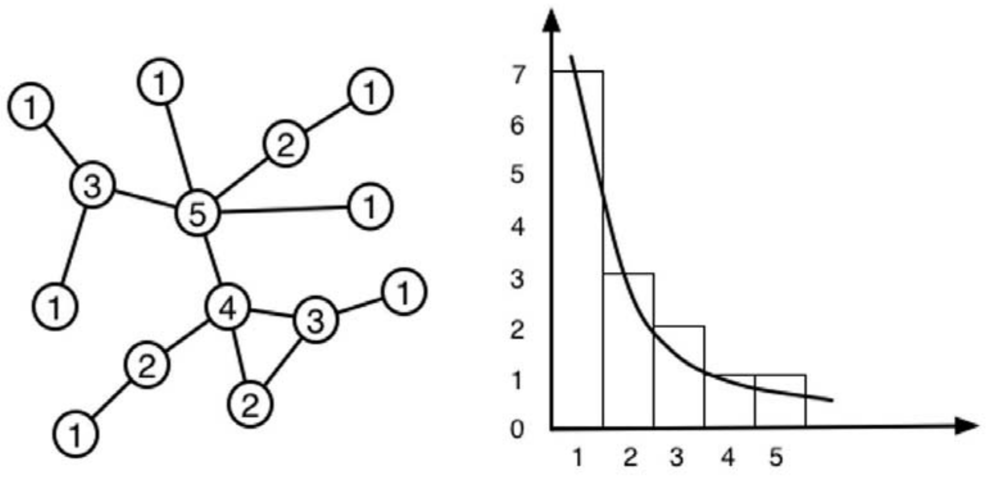

Fig. 1.4 The way in which the degree sequence is computed and plotted by means of a histogram. Node labels are the degrees.

Histogram

In Python it is extremely easy to plot any kind of graphs and one of the most popular libraries is Matplotlib. In order to get the histogram of the previous degree sequence we simply issue:

```
import matplotlib.pyplot as plt

plt.hist([1,1,1,1,1,1,1,2,2,2,3,3,4,5],bins=5)
plt.show()
```

1.5 Clustering coefficient and motifs

The *clustering coefficient* is a standard, basic measure of the community structure at local scale. Imagine a network of friendship (visualised as edges) between persons (vertices). The clustering coefficient gives the probability that if Frank is a friend of John and Charlie, also John and Charlie are friends with each other. For graphs this means that if we focus on a specific vertex i connected to other vertices, the clustering coefficient c_i measures the probability that the destinations of these vertices are also joined by a link. If all the connections are equiprobable, we just count the frequency of such connections, that is, we measure the number of triangles insisting on a particular vertex, as shown in Fig. 1.5 a. Another measure used is the clustering coefficient $c(k)$ of vertices whose degree is k. This is the average of all the values of the clustering coefficients made with all the vertices whose degree is k,

$$c(k) = \frac{\sum_{i=1,N} c_i \delta_{k_i,k}}{N_k}, \tag{1.4}$$

where N_k is the number of vertices whose degree is k and $\delta_{k_i,k} = 1$ if $k_i = k$ and 0 otherwise. Real networks are often characterised by a clustering larger than expected

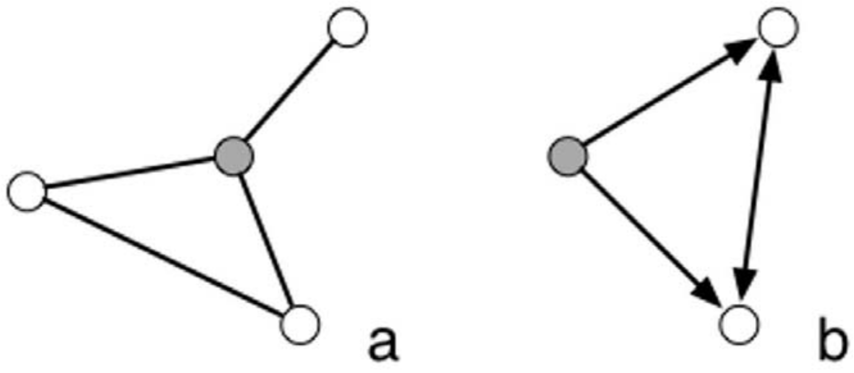

Fig. 1.5 (a) Case of a vertex (in grey) whose clustering is 1/3 (one triangle out of the possible three. (b) A feed-forward loop.

from a series of randomly connected vertices. It is worth noticing that in the case of directed networks as with food webs, it is somewhat difficult to determine which triangles must to be considered, since now the edges have a direction. By considering this further degree of freedom, a simple triangular structure can assume a variety (nine) of different shapes. Such shapes are called *motifs* and their statistics can reveal something about the density of the system at a local scale. For example, food webs are characterised by the net dominance of a "feed-forward" loop, as shown in Fig. 1.5 b. Similarly, motifs can be used to detect early signals of collapse in the particular case of financial networks (Squartini *et al.*, 2013).

Clustering coefficient

As for the node degree previously defined, we can code the clustering coefficient for a specific node. For example, looking at Fig. 1.5 and node "2" we can express the clustering coefficient computing the connections between the neighbours of node "2" and dividing by all the possible connections among them (degree*(degree-1)/2). First we compute the list indices of the neighbours of "2":

```
row=1 #stands for the node 2
node_index_count=0
node_index_list=[]
for a_ij in adjacency_matrix[row]:
    if a_ij==1:
        node_index_list.append(node_index_count)
    node_index_count=node_index_count+1
print "\r"

print node_index_list
```

and the list in the case of node "2" will be

```
#OUTPUT
[0, 2, 3]
```

then we will check all of the possible neighbour couplings for whether a link actually exists:

```
neighb_conn=0
for n1 in node_index_list:
    for n2 in node_index_list:
        if adjacency_matrix[n1][n2]==1:
            neighb_conn=neighb_conn+1

#we have indeed count them twice...
neighb_conn=neighb_conn/2.0

print neighb_conn

#OUTPUT
1.0
```

and in our case the result is simply 1. Finally the clustering coefficient for node "2" is given by the expression

```
clustering_coefficient=neighb_conn/ \
(degree_node_2*(degree_node_2-1)/2.0)

print clustering_coefficient
```

where the final result is 0.333333333333.

1.5.1 Ecological level and categories between species, bowtie

One of the distinctive features of food web data is the possibility of arranging the vertices along different levels defined by the distance from the environment (as usual, the distance in a graph corresponds to the minimum number of edges to travel between two vertices). As a result we can define categories according to the in/out links relating to the predation. All the species that have no predations are indicated as top (T), all the species with no prey (apart from the environment) are indicated as basal (B). All the others are intermediate (I). Apart from the basal species, the intermediate or top species can be more or less distant from the environment. Probably (but not necessarily!), species at the lowest level are likely to be basal, while species on the highest levels are likely to be top ones. The study of universality in number of levels and composition is one of the traditional quantitative ecological analysis in the quest for food web universality. In order to identify the various levels in the food web network we need an algorithm able to compute the distance between all pairs of nodes. A generalisation of this concept of levels and classes of nodes is given by the concept of bowtie, a structure that was first noticed in the World Wide Web (Broder *et al.*, 2000), and late in economics (Vitali *et al.*, 2011) and financial systems. In any directed network you can determine a set of nodes mutually reachable one from another. They form the strongly connected component (SCC). Those from which you arrive at SCC are the IN component. Those reachable from the SCC form the OUT component. In spite of the technical differences between top species and OUT components, the two structures have some similarities (see Fig. 1.6).

Calculating the bowtie structure for a food web network

```
#loading the network
file_name="./data/Ythan_Estuary.txt"

DG = nx.DiGraph()

in_file=open(file_name,'r')
while True:
    next_line=in_file.readline()
    if not next_line:
        break
    next_line_fields=next_line[:-2].split(' ')
    node_a=next_line_fields[1] #there is a space in the beginning
                               #of each edge
    node_b=next_line_fields[2]
    DG.add_edge(node_a, node_b)

#deleting the environment
DG.remove_node('0')

#getting the biggest strongly connected component
scc=[(len(c),c) for c in sorted( nx.strongly_connected_components \
                               (DG), key=len, reverse=True)][0][1]

#preparing the IN and OUT component
IN_component=[]
for n in scc:
    for s in DG.predecessors(n):
        if s in scc: continue
        if not s in IN_component:
            IN_component.append(s)

OUT_component=[]
for n in scc:
    for s in DG.successors(n):
        if s in scc: continue
        if not s in OUT_component:
            OUT_component.append(s)
```

```
#generating the subgraph
bowtie=list(scc)+IN_component+OUT_component
DG_bowtie = DG.subgraph(bowtie)

#defining the proper layout
pos={}
in_y=100.
pos['89']=(150.,in_y)

in_step=700.
for in_n in IN_component:
    pos[in_n]=(100.,in_y)
    in_y=in_y+in_step

out_y=100.
out_step=500.
for out_n in OUT_component:
    pos[out_n]=(200,out_y)
    out_y=out_y+out_step

pos['90']=(150.,out_y)

#plot the bowtie structure
nx.draw(DG_bowtie, pos, node_size=50)

nx.draw_networkx_nodes(DG_bowtie, pos, IN_component, \
                       node_size=100, node_color='Black')
nx.draw_networkx_nodes(DG_bowtie, pos, OUT_component, \
                       node_size=100, node_color='White')
nx.draw_networkx_nodes(DG_bowtie, pos, scc, \
                       node_size=200, node_color='Grey')

savefig('./data/bowtie.png',dpi=600)
```

The simplest algorithm to determine paths and distances is an exploration known as Breadth First Search (BFS).

Distance with Breadth First Search

As shown in Fig. 1.7 the strategy to compute the distance from the root node is to explore all the accessible neighbours not already visited.

```
#creating the graph
G=nx.Graph()
G.add_edges_from([('A','B'),('A','C'),('C','D'),('C','E'),('D','F'),
```

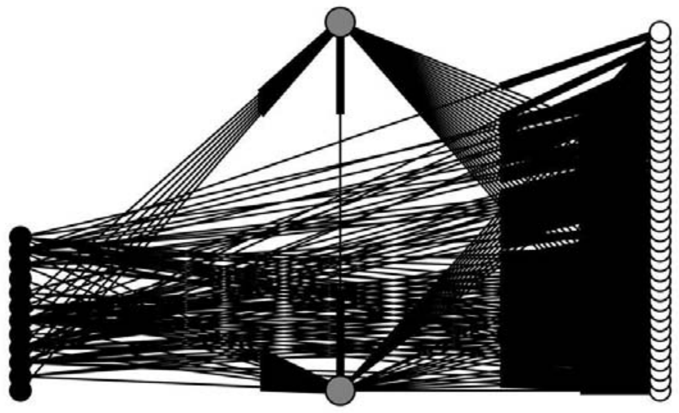

Fig. 1.6 A representation of a bowtie structure for the Ythan Estuary food web network. On the left the IN component in black. In the middle the two nodes of the strongly connected component in grey. On the right the OUT component in white.

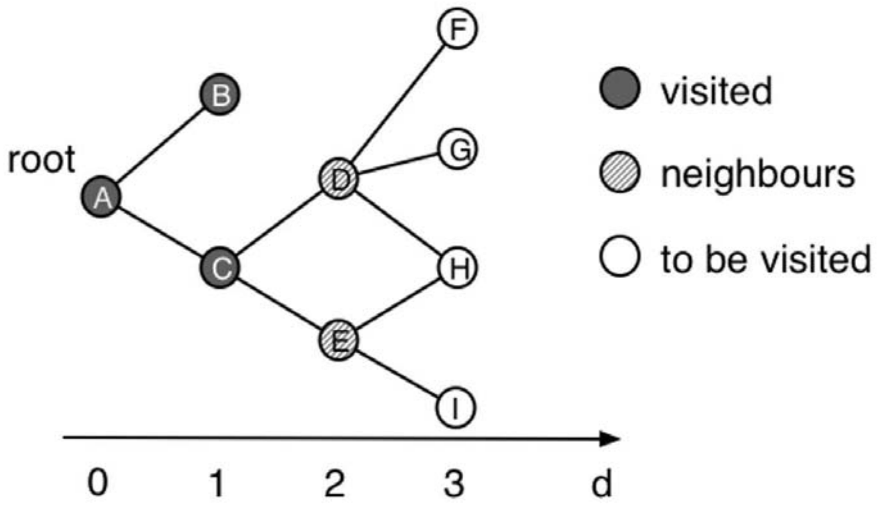

Fig. 1.7 The procedure of the Breadth First Search (BFS) algorithm. Starting from a root node at distance $d = 0$ at successive time steps we explore the neighbours at increasing distances.

```
('D','H'),('D','G'),('E','H'),('E','I')])

#printing the neighbors of the node 'A'
print G.neighbors('A')
```

```
#OUTPUT
['C', 'B']

root_node='A'
queue=[]
queue.append('A')
G.node['A']["distance"]=0
while len(queue):
    working_node=queue.pop(0)
    for n in G.neighbors(working_node):
        if len(G.node[n])==0:
            G.node[n]["distance"]=G.node[working_node]["distance"]+1
            queue.append(n)
for n in G.nodes():
    print n,G.node[n]["distance"]

#OUTPUT
A 0
C 1
B 1
E 2
D 2
G 3
F 3
I 3
H 3
```

As previously mentioned, we need trophic species to remove dimensional bias in the food webs; this is done by concentrating in a single node all the nodes with the same prey/predator pattern. In Table 1.1 we show trophic versions of the food web datasets we presented at the beginning of this chapter and their fundamental measures (Dunne *et al.*, 2002; Caldarelli *et al.*, 1998). In the following we will compute them, step by step, with simple Python code.

The first thing to do is to load the dataset in the shape of a network:

Reading the file with food web data

```
file_name="./data/Little_Rock_Lake.txt"

DG = nx.DiGraph()

in_file=open(file_name,'r')
while True:
    next_line=in_file.readline()
    if not next_line:
        break
    next_line_fields=next_line[:-2].split(' ')
    node_a=next_line_fields[1] #there is a space in the beginning
                               #of each edge
    node_b=next_line_fields[2]
    print node_a,node_b
    DG.add_edge(node_a, node_b)

#OUTPUT
0 11
0 61
0 80
0 123
0 124
...
```

Once the specific food web has been loaded in the Networkx structure we can operate on it. The first thing to do is to generate trophic versions of this network. We will use extensively the property of the dictionary key to be a complex data structure. In the present case we will use the list/tuple as a pattern to identify the particular trophic species.

Defining the trophic pattern key

```
def get_node_key(node):
    out_list=[]
    for out_edge in DG.out_edges(node):
        out_list.append(out_edge[1])
    in_list=[]
    for in_edge in DG.in_edges(node):
        in_list.append(in_edge[0])
    out_list.sort()
    out_list.append('-')
    in_list.sort()
    out_list.extend(in_list)
    return out_list
```

Leveraging from this pattern function we can extract the trophic species through the following function:

Grouping the trophic species and regenerating the trophic network

```
def TrophicNetwork(DG):
    trophic={}
    for n in DG.nodes():
        k=tuple(get_node_key(n))
        if not trophic.has_key(k):
            trophic[k]=[]
        trophic[k].append(n)
    for specie in trophic.keys():
        if len(trophic[specie])>1:
            for n in trophic[specie][1:]:
                DG.remove_node(n)
    return DG

#deleting the environment
DG.remove_node('0')

TrophicDG=TrophicNetwork(DG)
print "S:",TrophicDG.number_of_nodes()
print "L:",TrophicDG.number_of_edges()
print "L/S:",float(TrophicDG.number_of_edges())/ \
TrophicDG.number_of_nodes()

#OUTPUT
S: 93
L: 1034
L/S: 11.1182795699
```

Categories in a food web network are in relation to in/out links of each species. We have the basal ones (B) that are the prey (outgoing links only), the top species T (ingoing links only) which are only predators, and finally the intermediate species I (with in- and outgoing links) which are both prey and predators. Here is the Python code that categorised the “Little Rock” food web network introduced before:

Classes in food webs

```
def compute_classes(DG):
    basal_species=[]
    top_species=[]
    intermediate_species=[]
    for n in DG.nodes():
        if DG.in_degree(n)==0:
            basal_species.append(n)
        elif DG.out_degree(n)==0:
            top_species.append(n)
        else:
            intermediate_species.append(n)
    return (basal_species,intermediate_species,top_species)

(B,I,T)=compute_classes(TrophicDG)
print "B:",float(len(B))/(len(B)+len(T)+len(I))
print "I:",float(len(I))/(len(B)+len(T)+len(I))
print "T:",float(len(T))/(len(B)+len(T)+len(I))

#OUTPUT
B: 0.129032258065
I: 0.860215053763
T: 0.010752688172
```

Finally, we compute the proportion of the links among the various classes previously defined and the ratio prey/predators, defined as: $(\#B + \#I)/(\#I + \#T)$.

Proportion of links among classes and ratio prey/predators

```
def InterclassLinkProportion(DG,C1,C2):
    count=0
    for n1 in C1:
        for n2 in C2:
            if DG.has_edge(n1,n2):
                count+=1
    return float(count)/DG.number_of_edges()

print "links in BT:",InterclassLinkProportion(TrophicDG,B,T)
print "links in BI:",InterclassLinkProportion(TrophicDG,B,I)
print "links in II:",InterclassLinkProportion(TrophicDG,I,I)
print "links in IT:",InterclassLinkProportion(TrophicDG,I,T)

#Ratio prey/predators
print "P/R:",float((len(B)+len(I)))/(len(I)+len(T))

#OUTPUT
links in BT: 0.000967117988395
links in BI: 0.0909090909091
```

Experimental data						
	Silwood	Grassland	St Marks	St Martin	Ythan	L. Rock
S	16	15	29	42	83	93
L	33	30	262	203	398	1034
L/S	2.0	2.0	9.0	4.8	4.8	11.1
B (%)	21	13	10	14	5	13
I (%)	49	74	90	69	59	86
T (%)	30	13	0	17	36	1
TB (%)	10	3	0	3	1	0
IB (%)	29	10	13	19	10	9
II (%)	29	57	87	53	51	91
TI (%)	32	30	0	25	38	0
P/R	0.89	1.0	1.11	0.97	0.67	1.13

Table 1.1 Some basic quantities in various food webs. The data come from the following publications St. Martin (Goldwasser and Roughgarden, 1993), Ythan (Hall and Raffaelli, 1991), Little Rock (Martinez, 1991).

```
links in II: 0.908123791103
links in IT: 0.0
P/R: 1.13580246914
```